Design, Distribution and Management of Object-Oriented Software

Arindam Banerji, David Cohn, Dinesh Kulkarni
Distributed Computing Research Laboratory
University of Notre Dame
Notre Dame, IN 46556
axb@cse.nd.edu

Abstract

The promise of object-oriented software has been somewhat dimmed by the continuing need for source code familiarity to realize the goals of code-reuse and manageability. Software design has been hampered by the infeasibility of predicting all possible circumstances of use. Composing applications out of reusable components has remained a myth, primarily due to limitations of the simplistic shared library model. This paper proposes a three-pronged attack on these limitations of object-oriented software in the context of C++. A flexibility framework which facilitates the extension and modification of software without recompilations or source-code familiarity is described. Partially resolved loadable subclasses that may be distributed as reusable units for type-safe application composition. Specific programming guidelines which allow implementors to create software that may be fine-tuned at run-time according to application characteristics is described. Thus, the paper proposes a set of tools, techniques and guidelines that can facilitate the construction of application frameworks.

1. Introduction

Object-orientation has not lived up to its promise of promoting code-reuse and simplifying maintenance. Software ICs [Cox, 86] that were touted as solutions for expediting the development and maintenance of software, have failed to materialize. Here, we discuss some of the problems related to this issue and explore a possible solution in the context of C++, the most widely used object-oriented language.

We feel that a major roadblock on the way to easy maintenance and reusability, is the lack of *extensibility.* The designer and the initial implementor of a class library cannot predict *all* possible future uses. Hence, modifications and extensions of existing software typically requires an in-depth familiarity with its source code. However, if some of the normally *hidden* aspects of a software component's design and implementation are exposed in a disciplined fashion, the component can be extended without dependence on source code familiarity. We contend that inclusion of these facilities for extension and flexibility should be a basic design principle for object-oriented software.

Flexibility refers to the ability of a client of a software to tune the implementation of an object to suit client-specific needs. For example, adjusting the buffer size in a communication protocol implementation or changing the paging policy from least-recently used to most-recently used (for sequential reads) are examples of flexibility at work. Extensibility on the other hand entails major behavioral modifications; e.g. replacing a protocol implementation with a better implmentation that provides additional functionality by adding methods to the previous interface. It is important to emphasize that these two properties - flexibility and extensibility should be available at run-time, not just design and implementation time.

Extensibility would simplify distribution of class libraries. The need for releasing source-code would be substantially reduced; binary distribution would be sufficient in most cases. In addition, extensibility would make it possible for a user to dynamically integrate third party extensions to a particular product, thereby composing a tailored version of a standard application. This will safeguard the software vendor's investment without constraining clients who want to customize or enhance software. Moreover, it allows third-party vendors to sell customized extensions, much like Software ICs.

Flexibility and extensibility can be envisioned at several levels. Replacing a part or all of an implementation, binding to a different interface or even extending an interface, are all desirable for binary distribution without unduly constraining the client of the libraries. For example, a comprehensive communication software could choose between TCP/IP or NetBIOS interfaces, replace a UDP implementation or even extend the socket interface to support mobile communication. On another level, basic support technology for run-time subclass loading allows for dynamic extensions to existing software. In fact facilities for making such changes can be neatly organized into a framework.

Frameworks have been used as common units of reusable software [Deutsch, 89] [Wirfs-Brock, 90]. A framework is a set of collaborating class hierarchies that provides a certain Application Programming Interface (API) and uses a certain service-provider interface (SPI). For such application frameworks, extensibility could mean using a variant of the SPI, providing a variant of the API or modification of behavior and/or implementation. But there is a common thread that runs across these seemingly different aspects of flexibility. This thread can be captured in a framework for extensibility and flexibility that provides the basic mechanisms and a structure for tailoring an application framework. This framework forms the basis for application or domain-specific frameworks, and is the focus of this paper.

The proposed framework adds structure beyond the essential features of encapsulation and inheritance. It changes the black-box nature of components by providing facilities for exposing their implementation and for dynamically changing their behavior in an incremental fashion. It does so through a clean separation between an interface and its implementations on the one hand, and the interface and an associated instance on the other. It also uses systematic indirection for flexible binding and dispatch of alternate method implementations. A prototype of the basic components of the framework has been developed in AIX 3.2 using a custom variant of AT&T C++ translator. Sophisticated dynamic linking facilities constructed using AIX linker, allow for loading and binding of subclasses to running C++ programs.

The next section further explains the basic mechanisms for flexibility. It is followed by a section on programmer's view of extensible software and guidelines for extensibility. Section 5 discusses a programmer's view of a prototype implementation. Section 6 reviews some of the related work.

2. A Framework for Extensibility and Flexibility

As indicated in the preceding section, extensibility and flexibility must be supported from ground up rather than as add-on features or ad-hoc facilities in a particular application. This requires that the basic object model should itself address these issues and provide primitive facilities in the form of a generic framework which can then be used in different applications.

2.1. Basic Object Model

The basic object model, described in [Banerji, 93], envisages a clear separation between instances, interfaces and implementations as shown in Figure 1. The instance-interface separation allows multiple instances of an interface to change their behavior independently, and, since interfaces are explicit entities, they can be changed.

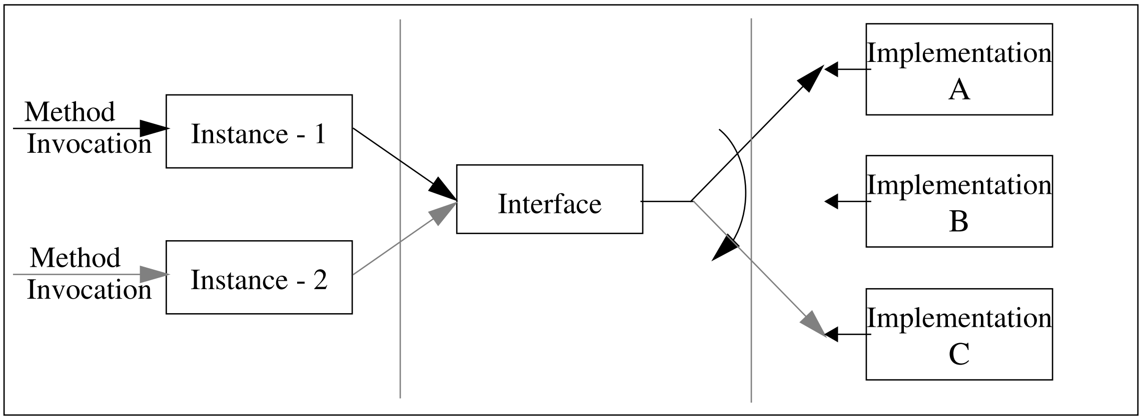

Figure 1 - Separating Instances, Interface and Implementations

Separation of interface from implementations ensures that multiple implementations for a particular interface can co-exist. This allows a client to choose a particular implementation at run-time. Initial implementors would build at least a default implementation, and users could add additional ones. One of the goals of such a separation is to ensure that classes representing interfaces depend upon very few symbols from classes representing implementations. This is especially helpful when implementations are loaded dynamically (as will be seen later) and symbol-binding has to be performed.

The initial implementation of this object model, as realized by the extensibility framework, currently provides only the interface-implementation separation. The mechanisms for providing this kind of a separation are discussed in Section 5 on programming guidelines.

2.2. Framework Components and Techniques

The framework utilizes indirection as a key mechanism for providing flexibility. Indirection in function dispatch ensures that implicit entities such as name-resolution and member function lookup can be directly controlled by both initial implementors and client-programmers. This control is usually realized in two forms. Function-names not known at compile time can become call-targets, allowing functions and classes that are loaded at run-time to be used directly. Preprocessing and postprocessing code can be dynamically attached to any member function, permitting the addition or deletion of pre-condition and post-condition functions on the fly.

Indirections are typically hosted in *class-objects* [Goldberg, 89]. These objects may be associated with any class for which extensibility is desired. Class-objects, also known as *meta-objects*, provide a run-time representation of normally implicit class implementation aspects. Implementors can construct class objects by specializing the facilities provided by the framework. The relationship between the implementor-constructed class-objects and the framework components is shown in Figure 2.

The upper part of the figure shows the set of collaborating class-hierarchies that are the components of the framework. These are combined by multiple-inheritance into the class `BaseMeta`.

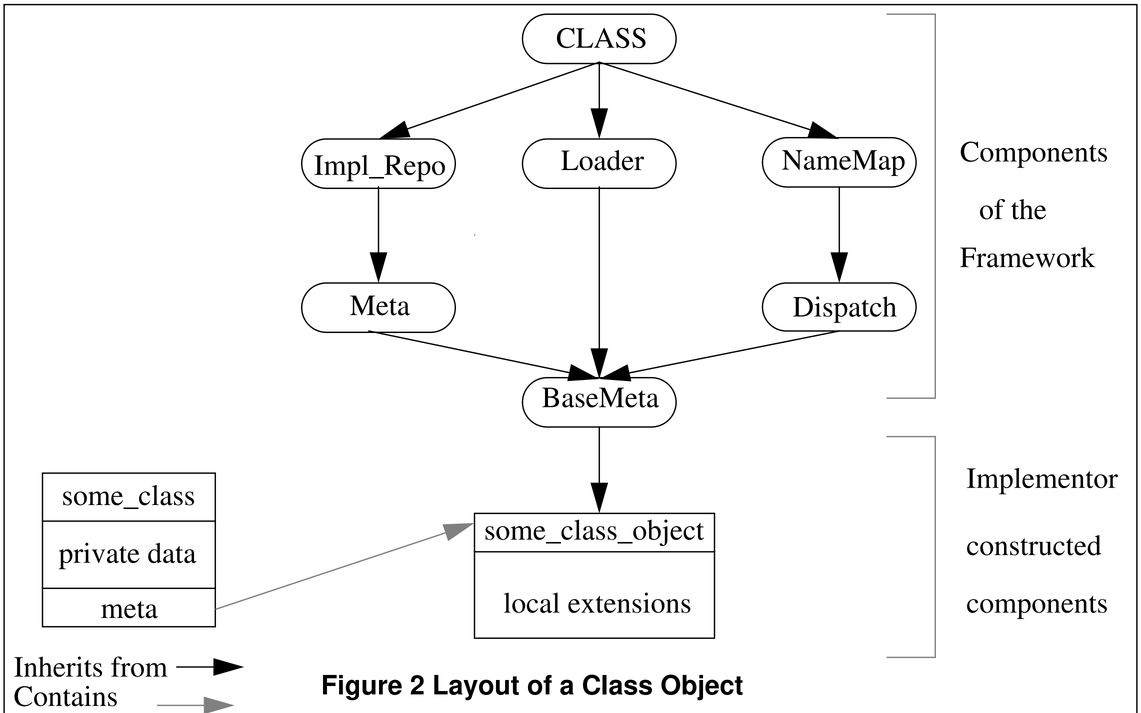

Figure 2 Layout of a Class Object

Class objects are then direct descendents of the class `BaseMeta.` They thus inherit the indirections provided by the framework.

These indirections are of two kinds: constructor-call and dynamic-dispatch. Constructor-call indirections allow the instantiation of classes whose names are not known at compile time. Thus, dynamically loaded classes identified by some characteristic string, such as their name, may be instantiated by client-code. Dynamic-dispatch indirection, on the other hand, allows the target of a member-function call to be derived from a variety of parameters and conditions, not just from the type-specification of the function. This allows clients code to manipulate exactly how the target of a named function call is resolved. Constructor-call indirection allows dynamic loading of new function implementations and dynamic-dispatch indirection allows clients to determine when these implementations are used.

These two kinds of indirection can be combined to facilitate the management of object-oriented libraries in two very special ways:

- Dynamically add or change implementations for a particular interface.
- Dynamically extend the interface provided by a library.

These changes can be made *without access to the source code*, providing true software reusability.

2.3. Constructor-Call Indirection

Dynamically loaded classes, enabled by indirections in constructor calling represent a unique problem since their names and the names of their member functions may not be known at compile-time. One approach is to only dynamically load subclasses of existing classes with matching function names. These functions must have initially been declared virtual and calls are then made through virtual function tables, ensuring type-safety. Of course, this limits new subclasses to using only previously known function names.

This still leaves the question of constructors, which in C++ cannot be declared virtual. Constructors of loadable subclasses need to be called indirectly to avoid unresolved externals during compilation. The classes **Meta** and **Impl_Repo** in Figure 2 implement this functionality. The details of the class Meta are shown below in a sample code fragment. The class **Meta** essentially supports three pieces of functionality:

- Functions **register_class_obj** and **unregister_class_obj** that allow the class-object of a loaded subclass to announce itself as available or unavailable as a possible target of constructor-call indirection.
- Overloaded function **instantiate** that actually redirects constructor calls.
- An instance of **Impl_repo** that acts as a database of available subclasses which can be targets of constructor call indirection.

The following code fragment shows the key features of the class **Meta**.

```
    class Meta: public CLASS {
        public:
            // constructors and destructors
/*
    In the following code ImplSpec is used to specify some characteristic
    of the implementation. In the simplest case, it may simply be a string
    containing the name of a class.
*/
            virtual void register_class_object(ImplSpec *,...);
                // function that allows the addition of the class-object
                // of a subclass[1] to the repository.
            virtual void unregister_class_object(ImplSpec *,...);
                // removes the entry for the class-object of a sub-class
                // from the repository.
            virtual CLASS *instantiate(ImplSpec *);
                // an indirect constructor called with some form of a
                // of a specification of which particular subclass
                // needs to be instantiated.
        private:
            static Impl_Repo *repository;
        };      // specification of the Meta class.
```

As expected, loadable subclasses are associated with class-objects. Global instances of these class-objects allow for automatic registration with the class-object for the base class, through calls to static constructors. For example, if **foo** and **fooMeta** are respectively the superclass and its associated class object and **childof_foo** and **childof_fooMeta** are the derived class and its associated class object, then the following happens during static initialization:

```
    childof_fooMeta childof_foo::meta = new childof_fooMeta;
```

This call to the constructor of **childof_fooMeta** causes it to register itself with **fooMeta**, as the class object for **childof_foo**. At this point, when instantiation requests for **childof_foo** are sent to **fooMeta**, they are automatically forwarded to the **instantiate** in **childof_fooMeta**. This **instantiate** function, in turn calls the constructor of **childof_foo**. If there are additional parameters that need to be passed to the constructor of **childof_foo**, then the code gets more complicated, but the principle remains the same.

1. Subclass here refers to the subclass of the class that the class that Meta is associated with. This, if **foo** is associated with **fooMeta**, a specialization of **Meta**, then the subclass here refers to **childof_foo**, a class derived from **foo**.

2.4. Dynamic Dispatch Indirection

Quite often, the ability to add or substitute class implementations is not sufficient. Certain circumstances require the interfaces themselves to be changed. When recompilation is possible, inheritance is used to create a modified class. However, if recompilation is not an option, a mechanism for adding member functions to existing interface classes is required. The components **`Dispatch`** and **`NameMap`** provide this functionality. The main features of these components are:

- The functions **`add_intf`** and **`remove_intf`** which allow the addition of member functions to the interface.
- An instance of **`NameMap`** that maintains the list of all the member functions added in this way.
- The function **`generic_func`** which indirectly calls the newly added member functions.
- Utility functions such as **`add_post`** and **`add_pre`** that add preprocessing and postprocessing code to member function.

The implementor creates a set of function macros, which map newly added member function calls to calls to **`generic_func`**. This function, in turn calls the actual function in the implementation, ensuring that no unresolved externals are generated during compilation.

In addition to the infrastructure described above, the **`Loader`** class provides control over loading of new code. This class gives clients of extensible software libraries an interface to add implementations at run-time.

3. Loadable Subclasses and Run-Time Loading

Much of the flexibility and reusability of the work described in this paper is gained from the ability to dynamically load and bind code. This entails the construction of special shared libraries for subclasses and a dynamic loading library which implements much of the functionality provided by the class **`Loader`** in Figure 2. The description that follows is specific to AIX 3.2 on the RS/6000s and the AT&T cfront compiler.

In order to create the appropriate shared libraries, the following steps need to be taken:

- Compile the source for the loadable subclasses and their associated class objects.
- Create a list of all symbols that are to be exported by this library.
- Create a list of all the symbols imported by this library, that should actually be resolved at run-time.
- Generate uniquely named library initialization and termination code and compile them.
- Use custom cfront compiler driver to link object modules from first and fourth steps into a shared library that has uniquely named arrays for holding static constructors and destructors.

Almost all the steps are completely automated, through generic makefiles and specially designed tools. The implementor need only specify the names of the classes that are to be exported by the library so created. Everything else is completely automated.

The run-time loading library dynamically links in and loads C++ libraries into running programs. At present, it only supports the XCOFF executable file format of AIX 3.2. The interface supported by the library mimics the SUNOS run-time linker calls of **`dlopen, dlsym and dlclose`**. During the call to **`dlopen,`** all unresolved externals within the loadable library, created as shown above, are bound to the appropriate symbols of the running client program. The **`dlopen`** and **`dlclose`** calls also ensure that the entry-points for calling static constructors and destructors, get called automatically.

It is very important to point out the difference between loadable subclasses as discussed here and ordinary shared libraries. Loadable subclasses do not preclude the type-safety provided by the C++ language. Moreover, many interesting characteristics of C++ such as virtual functions and inheritance do span across dynamically loaded subclass components. The features provided by these linking facilities would enable third party vendors to sell value-added subclasses for existing products, without being forced to ship any of the original product's class libraries. This ameliorates licensing problems and opens up possibilities for true pluggable software ICs.

4. Programmer's View

Having described the design of the tools used to build manageable software, we now discuss the construction of the software libraries. An implementor (one who builds manageable software) must meet certain guidelines in order to develop manageable and reusable software. These guidelines, along with some new approaches to object-oriented design lead to software more in tune with the needs of client code. In turn, programmers implementing clients of such software must be aware of the techniques for managing extending or fine-tuning the software. The next two subsections discuss these strategies from the perspectives of implementors of both the library as well as the client through an example of a socket library.

4.1. Implementor's View

The design of software libraries using the extensible framework is based upon three principles:

- Use of Dual-interfaces [Kiczales, 92b] to create open implementations
- Run-time representation of structure and behavior
- Separate interface and implementation hierarchies

Dual Interfaces represent a strategy based on recent research in language-design [Kiczales, 92a]. Traditionally, black-box abstractions have been used to hide implementation details. Conventional wisdom argues that this approach leads to more reusable and portable code. However, actual interfaces have often revealed implementation details for performance reasons. Implementors have cluttered up interfaces with functions that allow clients control over some aspect of the implementation. Violating encapsulation in this way often ensures that implementations can be fine-tuned by applications. The dual interface approach solves this dichotomy by defining two interfaces. A black-box interface presents the functionality of a software library to clients. An optional second *white-box* interface allows clients to control the implementation through hints and directives. Such an approach gives designers the luxury of basing implementation choices upon actual run-time usage patterns. Examples of choices are setting the message-buffer sizes for a `Socket` class or choosing the hardware link for message transmission.

Run-time manipulation of object behavior and structure requires that behavior and structure be represented in class definitions. As discussed above, behavioral representation in any class is enabled by class-objects accessed through the pointer `meta_obj`, a pointer to the meta-object. Structural representation on the other hand, would be enabled through the ANSI C++ RTTI extensions [Lajoie, 93]. Currently, a prototype library implementation of RTTI allows macro-based run-time type information to be embedded within a class. The declaration of a class needs to include a line of the form:

```
RTTI_SCAFFOLDING_DECL(Name_of_Class)
```

and the definition of the class needs to include a macro of the form:

```
RTTI_SCAFFOLDING_IMPL1(Name_of_Class, Name_of_Parent_Class)
```

Then, at run-time the type information pertaining to the class is available through the ANSI specified `get_info()` and the `info()` functions. This structural and type information is an identification mechanism for objects and enables run-time checks in dynamically loaded code.

The fragments of the interface and the implementation hierarchies, as shown in Figure 3, may now be examined. The interface hierarchy contains the primary interface and the dual interface. The primary interface consists of member functions that most UNIX programmers familiar with sockets would typically use. The dual interface adds the dimension of influence over implementation aspects of sockets, which are typically not under application programmer control. Clients only instantiate the objects that support the main (or primary) interface. The constructors of the main interface class automatically instantiate the dual interface object. The relationship between the main interface and the dual interface, is demonstrated through a hypothetical implementation of sockets, as shown below.

```
class Socket: public SocketBase {
    public:
            // constructors and destructors
            // Send a message
        virtual Socket &Send(...);
            // passes on certain calls to the second interface.
        Socket2ndInt *operator->();
            /* member functions for socket functionality*/
        ...
            // macro that generates RTTI scaffolding
        RTTI_SCAFFOLDING_DECL(Socket)
            // pointer to class object that controls
            // the behavioral metacomputation...
        static SocketMeta *meta;
    private:
                // Pointer to the optional second interface
        Socket2ndInt *SecondInterface;
};              // specification for the SocketInterface
```

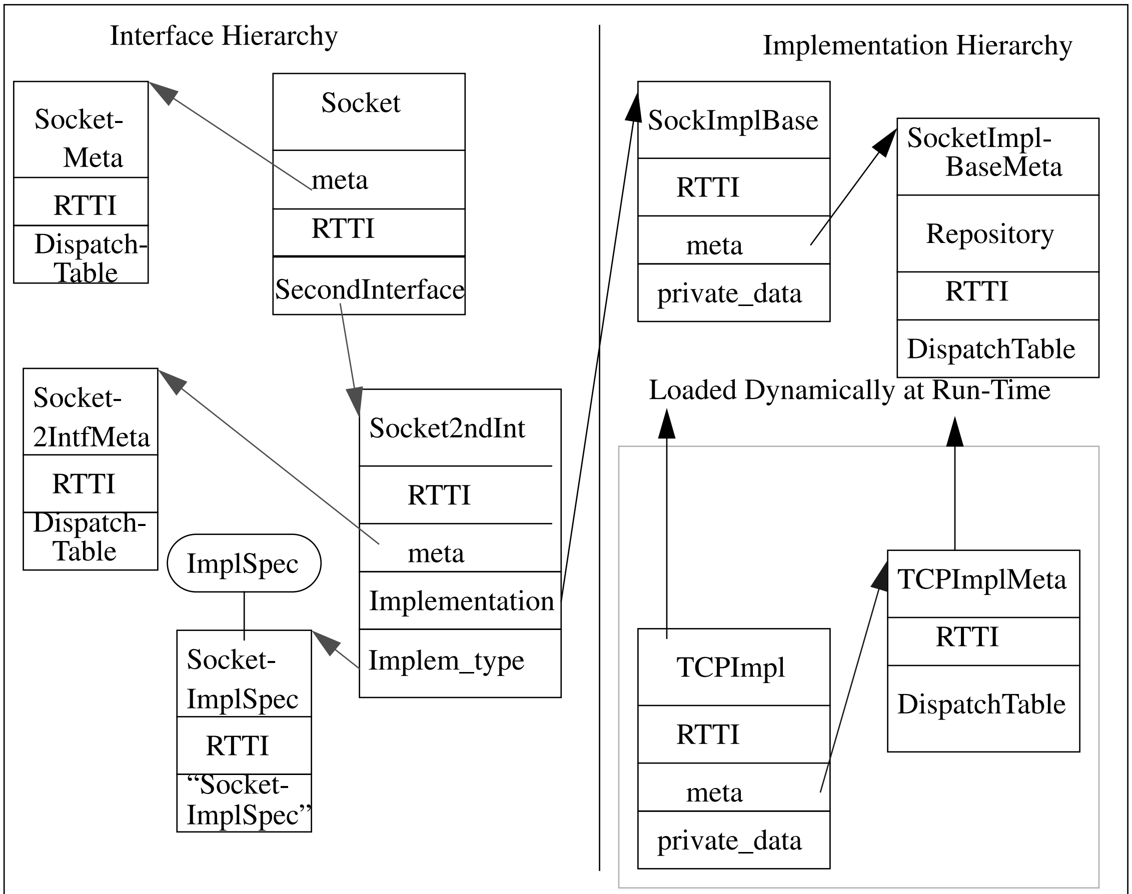

Figure 3. An Implementor's view of extensible software

The abstract root class of the implementation hierarchy (**SockImplBase**) and its class object are the only entities that are visible across the chasm between the two hierarchies. This allows the interaction between the two hierarchies to be limited to the member functions supported by **SockImplBase** and **SocketImplBaseMeta**. Thus, concrete implementations which inherit from **SockImplBase** are for the most part limited to the virtual member functions supported by **SockImplBase**. The only exception to this rule are dynamically added member functions that allow a particular implementation to support additional public member functions. Typically, concrete implementations of the **Socket** interface would inherit from **SockImplBase** and the class objects of these implementations inherit from **SocketImplBaseMeta**. Concrete implementations may be loaded at run-time or compiled in with the root of the implementation hierarchy.

Finally, in order to support multiple co-existing implementations, interface classes need to be able to identify specific implementations. This support is provided by the **ImplSpec** class hierarchy. **ImplSpec**, as shown below, defines a common protocol for comparison of implementation characteristics. Subclasses of the class **ImplSpec**, may use any characteristic such as the implementation-name to identify implementations, but have to support the comparison protocol defined by **ImplSpec**. Thus, typically every piece of extensible software defines its own subclass of **ImplSpec** in order to distinguish between different implementations.

```
// The following class defines the common comparison protocol, to be supp
// ported by all characterizations of implementations. Different exten
// sible software libraries may characterize implementations differ
// ently.Some may do it through implementation-name strings, while oth
// ers may use integer constants.

class ImplSpec : virtual public CLASS // CLASS is a global base, associ
// ated with properties, common to all objects of an extensible library.
     {
      public : // some operations have been omitted for brevity
         ImplSpec(const char *); // all characterizations are finally
                     // converted into strings, for simplicity.
         virtual ~ImplSpec() ; // destructor
         // The comparison protocol follows
         virtual int operator == (ImplSpec &) ;
         virtual int operator == (ImplSpec *) ;
         virtual int operator != (ImplSpec &) ;
         virtual int operator != (ImplSpec &) ;
         ... other comparison operators ...
     private :// mainly responsible for maintaining pointer to a
             // global repository, which catalogs all available
             // implementation characteristics.
         impl_map_t *table; // implementation repository front-end
         ...other private data...
     } ; // Base class of the Implementation Specification hierarchy
```

4.2. Client's View

A client of extensible software, thus engineered can use the interfaces provided for three purposes:

- Use the direct functionality of the interface e.g.: **Send**
- Use the second interface to control the implementation e.g.: **set_buffer_size**
- Use the meta-functionality to load new implementations of the interface.

The following piece of code demonstrates this for the socket class, discussed above. Initially, a client programmer allocates a socket object and selects the implementation to be used. For example, in this case the programmer decides to use the UDP protocol for the socket stream. After setting the implementation, the programmer may use the UDP socket as desired. At this point, for some hypothetical and fictitious reason, the programmer decides to use TCP, instead of UDP. Assuming, that the TCP implementation is not pre-loaded, the programmer asks for it to be loaded through the **add_impl** call. Subsequently, the TCP implementation is selected and the TCP socket is ready for use.

```
#include "socket.h"
    // Allocate a protocol object - the default implementation is
    // used - there may or may not be a default implementation.
    Socket *obj = new Socket;

    // Actually set the implementation to be used to be udp
    (*obj)->set_impl("UDP");
/* The pointer operator is used to get at the second-interface */

    // Use the functionality of the socketobject now.
    obj->Send(...);
            /* Some code here */...

    // At this point the client decides to add the tcp implementation
    // to the running program. It calls the interface provided by
    // the loader class, thru the meta pointer in the Socket class.
    obj->meta->add_impl("SocketImplBase","TCP");

    // Now the object may change the implementation type
    (*obj)->set_impl("TCP");
    // Now, the object can be called regularly, as in..
    (*obj)->set_buffer_size(...);
```

Occasionally, the client may want to load an implementation that extends the prescribed interface. Let us assume that the TCP socket is being used. At this point the programmer decides to use the MobileTCP stream, a realization of TCP that supports mobility of connections. Sockets of this stream support an additional member function **migrateconn**. One possible mechanism would be to create an new interface and use it to access the new implementation. However, sometimes recompilation of interface classes is not an option. In such cases, the programmer may dynamically extend the interface of the class, as shown below. The steps to be taken are as follows:

```
#include "socket.h"
#include "MobileTcpSocket.h"
    Socket *obj = new Socket;
    // Add the new implementation first
    obj->meta->add_impl("SocketImplBase", "MobileTCP");
    // Now set the implmentation to the MobileTcp type
    (*obj)->set_impl("MobileTCP");
    // Add the interface to the interface hierarchy - here the
    // name of the function to be added is "migrateconn"
    obj->meta->add_intf("SocketImplBase","MobileTCP","migrateconn");
    // Some code here to set up connections etc..
                ....
    // Migrate the connection by directly using the loaded function
    obj->migrate_conn(...);
```

Thus, clients can directly call the **migrateconn** function, as long as they are using the Mobile TCP implementation. Existing clients do not need to recompile any code. In order to avoid unresolved externals, the call to **migrateconn** gets converted to a call to **generic_func** at the preprocessing stage. Thus, clients that intend to use this new function must pull in the header files specific to this extended interface, so that the appropriate definitions of **migrate_conn** may be found.

5. Related Work

The need for extensibility in software has been stressed for both operating systems and languages [Kiczales, 92b]. The authors of the Meta-object protocol for CLOS [Kiczales, 91] have been instrumental in discussing open implementations and dual interfaces. Kiczales has also discussed extensibility specifically in the context of object-oriented libraries [Kiczales, 92a]. Lamping has elaborated the purpose and structure of the specialization interface for extensibility through inheritance [Lamping, 93].

Apertos (formerly called Muse) [Yokote, 92] has applied reflection and meta-object protocols to operating systems. The possibility of using frameworks for building reusable software was first discussed by Deutsch [Deutsch, 89]. Brad Cox [Cox, 86] coined the term Software ICs to refer to reusable components, primarily with respect to Objective-C. Choices [Campbell, 93], an object oriented operating system, supports frameworks for dynamic code loading and stepwise refinement. Recently, Open C++ [Chiba, 93] has used translator directives for redirecting method invocations to metaobjects to implement object groups in a distributed system. Aldus has used a custom-built preprocessor to add flexibility to their C++ application framework and to support scripting [Johnson, 93]. In comparison, the significance of this work lies in the structure and extent of extensibility provided in the context of a commercial operating system.

OMG's CORBA [OMG,91] and its implementations such as IBM-SOM [IBM,93] provide some basic facilities for separation of interface from the implementation and interoperability between multi-vendor components. However, our framework goes further by establishing a comprehensive set of metaobjects and providing a reference implementation. It provides facilities for adding to an interfaces, not just for changing an implementation.

The extensibility framework described in this paper may be thought of as a basis for concrete C++/Unix implementation of a set of reusable design patterns. The patterns are comparable to those observed by in ET++ and other similar frameworks [Gamma, 93].

6. Conclusion

The importance of making the management and modification of software cannot be overstressed. This project has taken some important steps in making Object-Oriented software more manageable and reusable. The framework for extensibility and the programming guidelines makes the task of implementors considerably easier. The use of dual-interfaces allows for better optimized implementations. Perhaps, the most important contribution is a step towards easing the distribution problems, facing the developers of object-oriented software components.

7. References

[Banerji, 93] A. Banerji et. al. The Substrate Object Model and Architecture, *Proc. IWOOOS '93*, pp. 31-41.

[Campbell, 93] R. Campbell et. al., Designing and Implementing Choices: An Object-Oriented System in C++, *Communications of the ACM*, 36(9), Sept, 93, pp. 117-126.

[Chiba, 93] S. Chiba & T. Masuda, Designing an Extensible Distributed Language with a Meta-Level Architecture, *ECOOP '93 - Object-Oriented Programming, LNCS 707*, Springer Verlag, pp. 482-501.

[Cox, 86] Cox, B.J, *Object-Oriented Programming - An Evolutionary Approach*, Addison Wesley, 1986.

[Deutsch, 89] Deutsch, L.P, Design Reuse and frameworks in the Smalltalk-80 programming system, *Software Reusability*, ACM Press, ACM Press, 1989.

[Firesmith, 93] D. Firesmith, Frameworks: The Golden Path to Object Nirvana, *Journal of Object-Oriented Programming*, 6(6), Oct. 93, pp. 6-8.

[Foote, 88] B. Foote & R. Johnson, Designing Reusable Classes, JOOP, 1(3),

[Gamma, 93] E. Gamma et. al., Design Patterns: Abstraction and Reuse of Object-Oriented Designs, *ECOOP '93 - Object-Oriented Programming, LNCS 707*, Springer Verlag, pp. 407-431.

[Goldberg, 89] A. Goldberg & D. Robson, *Smalltalk-80 The Language*, Addison Wesley, 1989.

[IBM, 93] SOMObjects Developer Toolkit User's Guide, Version 2.0, IBM.

[Johnson, 93] R. Johnson & M. Palaniappan, MetaFlex: A Flexible Metaclass Generator, *ECOOP '93 - Object-Oriented Programming, LNCS 707*, Springer Verlag, pp. 502-527.

[Kiczales, 91] G. Kiczales et. al., *The Art of Metaobject Protocol*, MIT Press.

[Kiczales, 92a] G. Kiczales & J. Lamping, Issues in the Design and Documentation of Class Libraries, Proc. OOPSLA '92, ACM, pp. 435-451.

[Kiczales, 92b] G. Kiczales, Towards a New Model of Abstraction in the Engineering of Software, *Proc. Workshop on Reflection and Meta-level Architectures, IMSA '92*.

[Lajoie, 93] H. Lajoie, Standard C++ Update - The New Language Extensions, *C++ Report,* July-Aug. 93, pp. 47-52.

[Lamping, 93] J. Lamping, Typing the Specialization Interface, *Proc. OOPSLA '93*, ACM, pp. 201-214.

[OMG, 91] The Common Object Request Broker: Architecture and Specification, OMG Document No. 91.12.1, Rev. 1.1, Object Management Group, Framingham, MA.

[USL, 92] *C++ Language System*, Unix System Labs.

[Wirfs-Brock, 90] R. Wirfs-Brock, R. Johnson, A Survey of Current Research in Object-Oriented Design, *Commun. of the ACM,* 33(9), pp. 104-124.

[Yokote, 92] Y. Yokote, The Apertos Reflective Operating System: The Concept and its Implementation, *Proc. OOPSLA '92*, ACM, pp. 414-434.